High-Frequency Absorption-FIFO Pipelining for Stratix 10 HyperFlex

Authors removed for blind review

***Abstract*—FPGAs commonly have significantly lower clock frequencies than many microprocessors and GPUs, due largely to propagation delays incurred by the reconfigurable interconnect. The Stratix 10 HyperFlex architecture reduces this problem by embedding numerous registers throughout the routing resources. However, such Hyper-Registers do not support back-pressure (i.e., pipeline stalls) that is commonly used in FPGA pipelines. In this paper, we present and evaluate pipeline transformations using absorption FIFOs, which avoid back-pressure limitations to enable numerous pipelines to benefit from HyperFlex, while also eliminating potentially expensive stall penalties incurred by existing techniques. We demonstrate that these transformations not only enable significant clock improvements on Stratix 10, but also for devices without HyperFlex, potentially making absorption FIFOs a better high-frequency strategy for any FPGA. In addition, we introduce optimizations that yield additional performance improvements by reducing stall penalties that can increase linearly with pipeline depth when restarting after a stall.**

Keywords-FPGA; Stratix 10; HyperFlex; pipelining

I. INTRODUCTION

Field-programmable gate arrays (FPGAs) provide massive amounts of parallelism [1], but performance for many applications is limited by clock frequencies that are often more than an order of magnitude slower than microprocessors and graphics-processing units. A significant contributor to slow clocks is propagation delay incurred by the reconfigurable interconnect. The Stratix 10 HyperFlex architecture [2] reduces this problem by embedding numerous flip-flops, referred to as Hyper-Registers, across the routing resources. These Hyper-Registers provide an optional pipelined interconnect, which has been shown to significantly improve clock frequencies [4].

One limitation of Hyper-Registers is the lack of support for back-pressure (i.e., pipeline stalls). Back-pressure is required by any streaming circuit where a downstream component must tell upstream components to stop sending data. For example, a DMA interface writing to an external DRAM must tell a pipeline to stall while the memory refreshes. As shown in Figure 1(a), FPGA pipelines often provide back-pressure using an enable signal connected to all registers. Because Hyper-Registers lack enable signals, synthesis maps this implementation onto normal registers, preventing any benefit from the pipelined interconnect.

Figure 1(b) illustrates a pipeline transformation [3] that provides the illusion of back-pressure without an enable by adding a FIFO on the pipeline output, which we refer to as an *absorption FIFO*. By providing an almost-full flag that leaves sufficient room to *absorb* the full contents of the pipeline, this FIFO can inform upstream components to stop sending without losing any data. However, one potential disadvantage of absorption FIFOs is that pipelines can become empty during a stall, which may increase stall penalties due to the FIFO emptying before the pipeline refills. For frequent stalls, this penalty is potentially prohibitive due to linear scaling with pipeline depth.

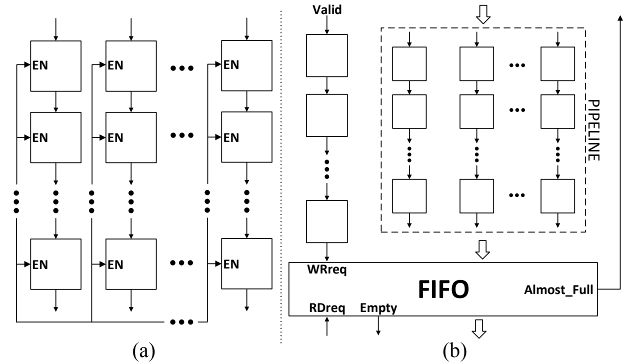


Figure 1: (a) Traditional FPGA pipelines often rely on back-pressure using high fan-out enable signals. In this paper, we demonstrate (b) pipeline transformations that eliminate the enable signal using an *absorption FIFO*, which enables use of Stratix 10 Hyper-Registers, while also reducing stall penalties and improving performance on older FPGAs.

This paper provides the following contributions. We first evaluate absorption FIFOs as an optimization for Stratix 10 HyperFlex, showing average clock frequency improvements of 17.5% compared to designs using back-pressure, with a maximum improvement of 54%. We also demonstrate that even for an Arria 10 without Hyper-Registers, the elimination of high fan-out enable signals provides average improvements of 8.5% and a maximum improvement of 41%, potentially making absorption FIFOs an overall better pipelining strategy for high frequencies. Finally, we introduce several optimizations and guidelines that ensure that absorption FIFOs can provide identical stall penalties as traditional pipelines.

II. RELATED WORK

Lewis et al. [4] presented a related study that adapted existing designs to Stratix 10. Our work complements that study with a more general optimization that can be applied to potentially any pipeline, and also evaluates the performance on published, as opposed to confidential, applications.

[3] describes numerous optimization strategies for Stratix 10, including a optimization similar to the presented absorption-FIFO strategy. Our work expands that description with more details on optimizations to eliminate stall penalties, while also evaluating improvements.

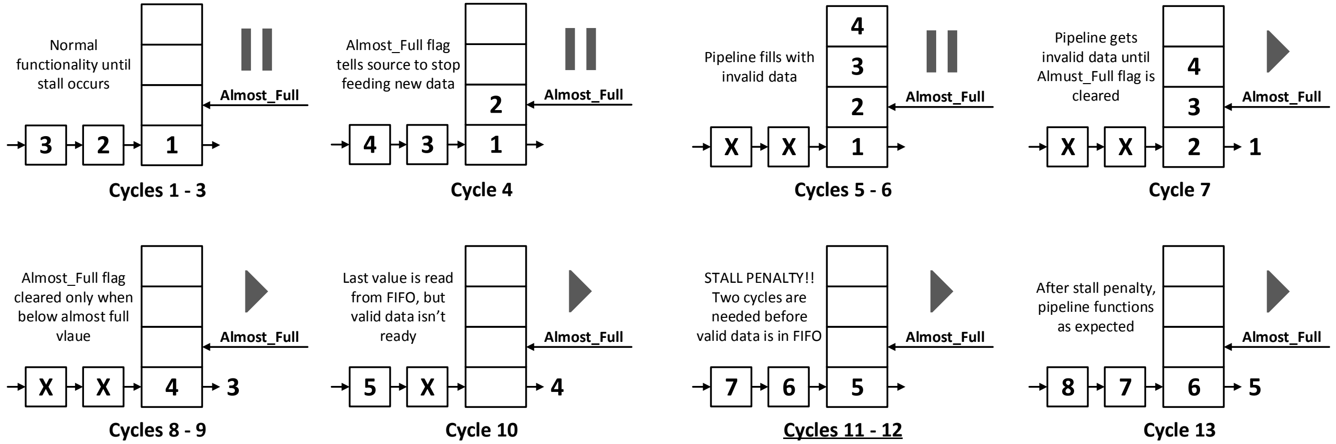


Figure 2: A demonstration of increased stall penalties for absorption FIFOs. Initially, the absorption FIFO functions like any other FIFO until almost full is asserted (Cycles 1-4). At that point, the absorption FIFO stops the sender while using reserved space to *absorb* the contents of the pipeline (Cycles 5-6). When a downstream component reads from the FIFO (Cycle 7), the pipeline remains empty until the FIFO clears the almost-full flag (Cycle 8). However, because the pipeline has not been refilled before the FIFO becomes empty, there are several cycles (Cycles 11-12), where the FIFO provides no output.

A recent study compared Stratix 10 with GPUs for deep neural networks, showing significant advantages for the FPGA [5]. Our study complements that paper with more general pipelining optimizations.

III. Absorption-FIFO Pipelining Strategies

In Section III.A, we first define how to use absorption FIFOs to eliminate back-pressure. Section III.B explains how absorption FIFOs can potentially have prohibitive stall penalties. In Section III.C, we present optimizations that eliminate this stall penalty.

A. Absorption FIFOs

Figure 1(a) shows a traditional pipelining strategy where a single enable signal is fanned-out to each register, allowing downstream components to immediately stop the progression of data until they are able to accept data again.

In this paper, we show that this traditional strategy has two significant limitations: 1) the high fan-out from the enable signal becomes a timing bottleneck for many pipelines, and 2) Hyper-Registers do not support enable signals, which significantly reduces clock frequency by preventing usage of the pipelined interconnect.

To solve both of these problems, pipelines must be transformed to eliminate the enable. Figure 1(b) illustrates an absorption-FIFO strategy that accomplishes this goal. With this strategy, whenever a component requests a stall, the upstream sender stops, but instead of stopping the pipeline, the absorption FIFO allows the pipeline to run continuously by absorbing the contents at the time of the stall.

Since the pipeline runs continuously, this strategy needs a mechanism to identify valid pipeline outputs. The strategy accomplishes this goal by including a shift register that delays a valid bit (asserted by the sender) by a number of cycles equal to the depth of the pipeline. The output of that shift register stores the pipeline output into the FIFO. Like other FIFO strategies, the FIFO's empty flag is used by the consumer to know when valid data is available to read.

The primary difference between a normal FIFO and the absorption FIFO is how the full flag is handled. Whereas a normal FIFO uses a full flag that is intended to immediately stop an upstream sender, an absorption FIFO uses an almost-full flag that allows for a delayed response from the sender.

To ensure there is enough space to absorb the contents of the pipeline, the minimum number of FIFO words is:

$$Min\ FIFO\ words = pipeline\ depth + 1$$

where the addition by one ensures that the almost flag is not continually asserted. Because this equation can lead to non-powers of two, for most practical situations where the FIFO is implemented in RAM, the minimum number of RAM words is defined by the following equation:

$$Min\ RAM\ words = 2^{ceil(\log_2(pipeline\ depth\ +\ 1))}$$

Regardless of the size of the FIFO used, the absorption FIFO requires the almost-full flag to be asserted when the number of words in the FIFO reaches:

$$almost\ full\ count = FIFO\ size - pipeline\ depth$$

For situations where a designer is already using a FIFO in a pipeline, this FIFO can be converted into an absorption FIFO by increasing the size by the pipeline depth and then using an almost-full flag as described above.

B. Stall Penalty Problem Definition

Depending on the length of the pipeline and the size of the FIFO, the absorption-FIFO strategy can lead to significant performance decreases upon resuming from a stall, which we refer to as the *stall penalty*.

Figure 2 demonstrates this problem by stepping through the contents of a pipeline and absorption FIFO. For simplicity of illustration, we use a 2-stage pipeline, and an absorption FIFO with four words, where the almost-full flag is asserted at word 2. This example assumes that an upstream sender (not shown) continuously produces data every cycle, as would typically occur in a pipeline. The example also assumes a downstream component (not shown), such as a

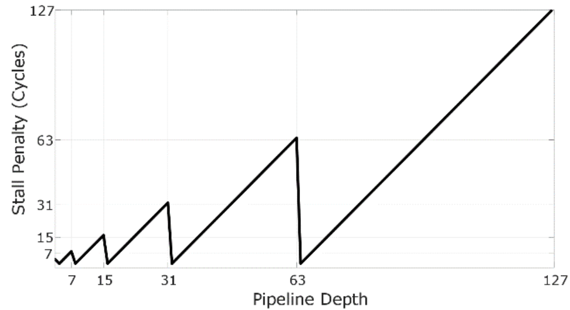


Figure 3: An illustration of maximum stall penalties for different pipeline depths. These results show that without the presented optimizations, stall penalties can increase linearly with pipeline depth.

memory, which reads data from the pipeline every cycle that data is available until it requests a stall.

For this example, the consumer stalls until Cycle 7 (shown as the pause symbol), which prevents reading from the FIFO. During Cycles 1-3, the sender pushes Elements 1-3 into the pipeline. In Cycle 3, Element 1 enters the FIFO, while Elements 2 and 3 remain in the pipeline. During Cycle 4, the FIFO stores Element 2, which asserts the almost-full flag, stalling the sender on the next cycle. Prior to receiving the stall, the sender provides Element 4 to the pipeline. During Cycles 5-6, the FIFO absorbs the contents of the pipeline (Elements 3-4) into the reserved space. By Cycle 6, the sender has been stalled for two cycles, which results in an empty pipeline with invalid data.

In Cycle 7, the downstream consumer begins reading from the FIFO (shown by the play symbol). However, because the almost-full flag is still asserted, the sender does not yet provide new data, causing the pipeline to remain empty. In Cycle 8, the consumer continues to read from the FIFO. In Cycle 9, the FIFO clears the almost-full flag, which allows the sender to resume. In Cycle 10, the consumer continues to read from the FIFO, while the sender begins to refill the pipeline.

The stall penalty is illustrated in Cycles 11-12. During this period, the consumer has emptied the FIFO, but the pipeline has not yet been refilled. As a result, the FIFO has a two-cycle stall penalty where the consumer does not receive new data. Finally, in Cycle 13, the pipeline has been refilled and the FIFO resumes outputting data to the consumer.

Figure 3 shows how the maximum stall penalty scales with the depth (number of stages) in the pipeline. In general, the penalty grows linearly with pipeline depth. For every power of two depth, the penalty decreases to 2 due to the FIFO doubling in size. At that point, 1 cycle is a result of having to go below the almost-full value and the other comes from not being able to store and read the same data in the same cycle. A fall-through FIFO would reduce this penalty by 1 cycle, but may result in a reduced maximum frequency. In general, the maximum stall penalty is determined by:

$$Penalty = pipeline\ length - (almost\ full\ value - 2)$$

Note that not all instances of absorption FIFOs will incur the maximum penalty. In addition, if stalls rarely occur, such penalties may be insignificant. However, for deep pipelines that stall frequently, the stall penalties can become prohibitive due to the linear increase with pipeline depth.

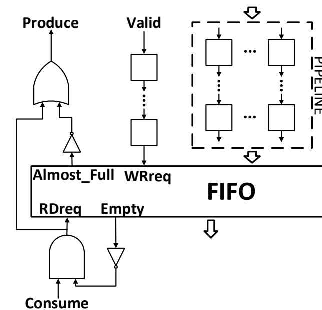


Figure 4: Modification to the absorption FIFO to eliminate stall penalties.

C. Optimizations

To avoid the stall penalties described in the previous section, we present two optimizations with various tradeoffs.

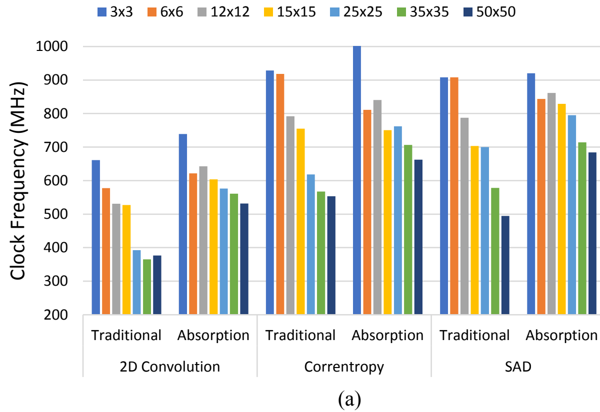
A low area-overhead optimization is to modify the logic used to stall the upstream producer, as shown in Figure 4. Instead of having the sender produce data solely when the almost-full flag is not asserted, this logic also has the sender produce data whenever the consumer reads data from the FIFO. This optimization guarantees sufficient room in the FIFO without the restriction of stalling the sender until the almost-full flag is cleared. For the example in Figure 2, this extension would enable the sender to resume in Cycle 7 as opposed to Cycle 10. The only limitation of this optimization is that it doesn't maintain the separation of parallel tasks normally provided by FIFOs, and as a result would not work for multiple clock domains without more logic.

An alternative optimization is to increase the FIFO size to provide sufficient buffering before the almost-full flag. At a minimum, this increased FIFO would require a number of words equal to the pipeline depth to buffer the entire pipeline both before and after the almost-full flag. Although a conceptually simple optimization, doubling the size of the FIFO for deep pipelines might not be a feasible option for designs that already use large amounts of RAM resources.

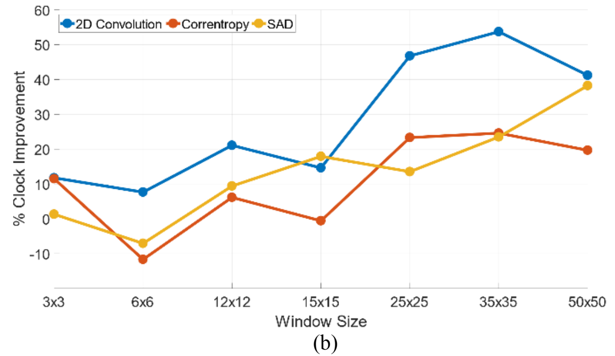
IV. Experiments

In this section, we compare clock frequencies of pipelines with and without absorption FIFOs. To evaluate both approaches, we implemented three optimized sliding-window pipelines from previous work [1]: SAD (sum of absolute differences), 2D convolution, and correntropy (a non-linear similarity measure). We picked sliding-window applications due to proven scalability limitations [1] that we can easily evaluate by varying window sizes, in addition to prevalence in important applications such as convolutional neural nets.

We evaluated the pipelines with and without absorption FIFOs on a Stratix 10 1SG280LN3F43E1VG using Altera Quartus 17.1 and an Arria 10 10AX016E4F29M3SG using Altera Quartus 16.0. We chose a smaller Arria 10 (61,510 ALMs) to better observe any fanout bottlenecks related to large pipeline sizes. We used different versions of Quartus for each device because we were unable to use virtual pins in



(a)



(b)

Figure 5: (a) A clock-frequency comparison of traditional pipelining and absorption FIFOs on Stratix 10 for sliding-window applications of different window sizes. (b) Percent improvement in clock frequency for the same experiments.

Quartus 17.1, which was required because the pipelines had thousands of inputs. Since Quartus 17.1 is the only version to support Stratix 10, we compensated by serializing the inputs using a shift register on a separate clock domain. Quartus 16 allowed us to use virtual pins for all Arria 10 examples.

To obtain clock frequency, we created a TCL script that varied clock constraints to find the maximum clock frequency reported by Quartus after placement and routing.

Figure 5(a) compares the clock frequencies on the Stratix 10 for the sliding-window pipelines. On average across all examples and window sizes, absorption FIFOs achieved a clock frequency of 737 MHz, compared to the average of 650 MHz with traditional pipelining. In general, as the pipeline sizes increased, the clock frequency decreased for both methods, as expected.

Figure 5(b) shows the same experiments using percent clock improvement to illustrate trends for different sizes. We see that in most instances, the absorption pipeline method outperformed the traditional method. The only exception was when traditional pipelining obtained frequencies near the maximum clock frequency of 1 GHz [2]. At this high frequency, the fanout was no longer the limiting factor, so the absorption method did not provide any benefit. On average, the improvement was 17.5%. While most pipeline sizes saw an improvement in clock frequency, larger pipeline sizes saw the most benefit. 2D convolution and correntropy achieved a maximum improvement of 54% and 25% at size 35x35, respectively, while SAD achieved a maximum 38% improvement at size 50x50. Beyond these sizes, the critical path shifted to other parts of the design, at which point absorption FIFOs provided less benefit.

Figure 6 shows absorption-FIFO clock improvements for the Arria 10, which were smaller than Stratix 10, but still significant for some cases. SAD achieved a 41% increase (a 155 MHz improvement) at size 15x15, which decreased for larger sizes. 2D convolution had modest improvements for small sizes, but improved to 31% (a 77 MHz improvement) for 24x24. For correntropy, absorption FIFOs provided no increase in frequency due to the critical path being unrelated to the fanout in the traditional approach.

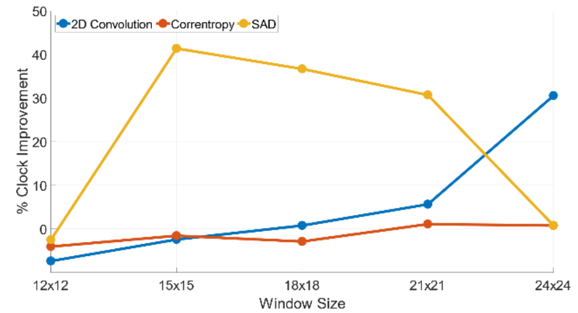


Figure 6: Percent improvement in clock frequency for absorption FIFOs on Arria 10 for sliding-window applications.

V. CONCLUSIONS

This paper demonstrated improvements to traditional back-pressure in pipelines by using absorption FIFOs. We showed that absorption FIFOs enable use of pipelined interconnect on Stratix 10 HyperFlex, providing average clock improvements of 17.5%, and a maximum of 54%, with a tendency to improve with larger pipelines. Absorption FIFOs also improved frequencies on an Arria 10 without Hyper-Registers, which makes the strategy a potentially better overall approach to high-frequency pipelining. Finally, we demonstrated optimizations to absorption FIFOs that eliminate potentially prohibitive stall penalties.

REFERENCES

[1] J. Fowers, G. Brown, P. Cooke, and G. Stitt, "A performance and energy comparison of fpgas, gpus, and multicores for sliding-window applications," in FPGA'12, pp. 47-56, 2012.

[2] M. Hutton, "Stratix 10: 14nm fpga delivering 1ghz," in 2015 IEEE Hot Chips 27 Symposium (HCS), pp. 1-24, Aug 2015.

[3] Intel Stratix 10 High-Performance Design Handbook. Jan. 2018.

[4] D. Lewis, G. Chiu, J. Chromczak, D. Galloway, B. Gamsa, V. Manohararajah, I. Milton, T. Vanderhoek, and J. Van Dyken, "The Stratix™10 highly pipelined fpga architecture," in FPGA'16, pp. 159-168, 2016.

[5] E. Nurvitadhi, G. Venkatesh, J. Sim, D. Marr, R. Huang, J. Ong Gee Hock, Y. T. Liew, K. Srivatsan, D. Moss, S. Subhaschandra, and G. Boudoukh, "Can fpgas beat gpus in accelerating next-generation deep neural networks?," in FPGA'17, pp. 5-14, 2017.